On the Equivalence of Probabilistic Automata: A Brief Literature Review

Abhinav Aggarwal
University of New Mexico, Albuquerque, NM USA

1 Introduction

Concepts as fundamental as formal languages and automata arise in day-to-day applications in Computer Science. Be it using the pushdown-automata for parsing programming languages or performing pattern matching on passwords and encoded strings using finite state machines, these simple structures provide elegant solutions in a variety of applications, including the design of sequential circuits, modelling protocols and decision procedures for logical formalisms. At the heart of all this functionality lies non-determinism, a blessing in disguise. The fundamental concept of non-determinism - the ability of a computational engine to guess a path to a solution and then verify it - brings in a ton of extra computational power which is often not possible with purely deterministic machines. However, this may not be the case always. For finite automata, Rabin and Scott (1959) showed that non-determinism does not add computational power. because every non-deterministic finite automaton (NFA) can be converted to an equivalent deterministic finite automaton (DFA) using the subset construction. However, since the subset construction may increase the number of automaton states exponentially, even simple problems can become computationally difficult to solve if non-determinism is involved.

One of the basic problems in automata theory is the equivalence problem : *given two automata A and B, do they define the same language, that is,* $L(A) \stackrel{?}{=} L(B)$. For DFA, we have a linear time algorithm by Hopcraft and Karp (1971). For NFA, however, the minimization algorithm does not solve the equivalence problem, but computes the stronger notion of bisimilarity between automata [HR15]. The Wikipedia definition of bisimilarity, or equivalently, bisimulation is a binary relation between state transition systems, associating systems that behave in the same way in the sense that one system simulates the other and vice versa. Intuitively two systems are bisimilar if they match each other's moves. In this sense, each of the systems cannot be distinguished from the other by an observer. The textbook algorithm for NFA equivalence involves a subset construction to first obtain equivalent minimal DFAs, which in the worst case takes exponential time and then compare these two DFAs using already existing efficient algorithms for the same. Indeed, it is unlikely that there is a theoretically better solution, at least for NFAs with ε-transitions, as the equivalence problem for NFA is PSPACE-Hard, which was shown by Meyer and Stockmeyer (1972).

With the merging of ideas from probability theory and computer science, one way to deal with worst case exponential time algorithms is possible through randomization. In this regard, algorithms which take large amount of time on some inputs but run fast enough on others can be made to run in a computationally feasible amount of time either by allowing randomization of inputs or by taking decisions about the outcome randomly (but based on some preprocessing). The simplest example of such an application is the Quicksort algorithm, which in the worst case takes quadratic time for sorting an array, but only $\mathcal{O}(n \lg n)$ time in the expected case. With such algorithms is introduced the notion of probabilistic computation. Some formal models in this domain include the probabilistic finite state automata and probabilistic Turing machines. These machines, as opposed to their non-probabilistic counterparts, further generalize the notion of non-determinism and instead of either accepting or rejecting an input string, impart a probability distribution

over the set of all possible input strings. Nowadays, probabilistic automata, together with associated notions of refinement and equivalence, are widely used in automated verification and learning. The upcoming section enhances more on this topic and discusses the equivalence problem in this setting.

2 Probabilistic Automata

As explained in [AW92], a probabilistic automaton is formalized as a stochastic matrix [1] M together with an initial distribution π over the set of states. Intuitively, this is similar to an NFA except that the transitions take place with probabilities prescribed by M. To start the process, the machine chooses an initial state according to π and then at any given point after that, the machine is in some state i, and at the next time step moves to another state j out-putting some letter z with probability specified by $M(i, j, z)$. If one stops the machine at time step n, it ends in some state having generated a string of length n. In this way, a probabilistic automaton naturally defines a probability distribution over the set of strings of length n, for any particular n.

Definition 1. *A probabilistic automaton P is a quadruple $\langle S_P, \Sigma_P, \pi_P, M_P \rangle$ where S_P is a finite set of states, Σ_P is a finite alphabet, $\pi_P : S_P \to [0, 1]$ is a probability distribution over S_P, and $M_P : S_P \times S_P \times \Sigma_P \to [0, 1]$ is a stochastic matrix, i.e.*

$$\sum_{i \in S_P} \pi_P(i) = 1 \quad \text{and } \forall i \in S_P, \text{ we have } \sum_{j \in S_P, z \in \Sigma_P} M_P(i, j, z) = 1 \tag{1}$$

Each $\pi_P(i)$ is called an initial probability, and each $M_P(i, j, z)$ is called a transition probability. For any string $w = w_1 w_2 \dots w_n \in \Sigma_P^*$, the generation probability assigned on it by P is computed as follows.

$$P(w_1 w_2 \dots w_n) = \sum_{\langle i_0, i_1, \dots, i_n \rangle \in S_P^{n+1}} \left(\pi_P(i_0) \cdot \prod_{j=0}^{n-1} M_P(i_j, i_{j+1}, w_{j+1}) \right) \tag{2}$$

Thus, for any given length n, P defines a probability distribution over Σ_P^n.

Sometimes, another parameter $\eta_P : S_P \to [0, 1]$ is added to this tuple to account for the acceptance probability of each state. Intuitively, this corresponds to the probability that if parsing a string w ends in state i, then this state accepts w in the language with probability $\eta_F(i)$ and rejects it otherwise. This is a minor generalization over the existing definition, but will prove useful in developing the algorithm by Tzeng [Tze92] for a polynimial time equivalence check on two given probabilistic automata.

Let us take an example before we move further. The probability assigned by the probabilistic automaton P shown in Figure 1 on the string $w = aab$ is calculated as follows:

$$\begin{aligned} P(w) = {} & \pi_P(0) \cdot M_P(0, 0, a) \cdot M_P(0, 1, a) \cdot M_P(1, 1, b) \cdot \eta_F(1) \\ & + \pi_P(0) \cdot M_P(0, 1, a) \cdot M_P(1, 1, a) \cdot M_P(1, 1, b) \cdot \eta_F(1) \\ & + \pi_P(0) \cdot M_P(0, 1, a) \cdot M_P(0, 2, a) \cdot M_P(2, 0, b) \cdot \eta_F(0) \\ & + \pi_P(1) \cdot M_P(1, 1, a) \cdot M_P(1, 1, a) \cdot M_P(1, 1, b) \cdot \eta_F(1) \\ & + \pi_P(1) \cdot M_P(1, 1, a) \cdot M_P(1, 2, a) \cdot M_P(2, 0, b) \cdot \eta_F(0) \\ & = 0.1835 \end{aligned}$$

For the sake of notation, given a probabilistic automaton P, we will use the same letter P to denote the probability distribution defined by P on Σ_P^n, where n will be clear from the context. Now, given this

[1] A stochastic matrix M is one in which each entry lies in the interval $[0, 1]$ and the sum of all entries in each row is 1.

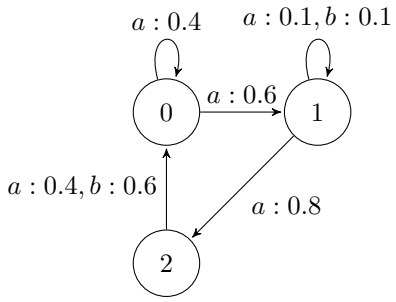

Figure 1: An example probabilistic automaton with $\pi_P(0) = 0.5, \pi_P(1) = 0.5, \pi_P(2) = 0, \eta_P(0) = \eta_P(1) = \eta_P(2) = 1$.

definition and intuition behind designing probabilistic automata, it becomes clear why they form a good model for probabilistic programs in general, atleast for the ones having only finitely many states. A very natural question then arises about the equivalence of two such automata. Deciding this equivalence is a key problem for establishing various behavorial and anonymity properties of probabilistic systems. However, the definition of equivalence must be made unambiguous before we delve into details. We borrow the definition from [KMO+12].

Definition 2. *Two probabilistic automata are said to be equivalent if each word is accepted with the same probability by both the automata.*

It was shown by Tzeng [Tze92] that equivalence for probabilistic automata is decidable in polynomial time. However, surprisingly, the natural analog of this problem, called the language inclusion (that one automaton accepts each word with probability at least as great as another automaton), is undecidable even for automata of fixed dimension. The paper [KMO+12] also points out that the equivalence problem for probabilistic automata can be solved by reducing it to the minimization problem for weighted automata and applying the algorithm by Schützenberger. However, the paper itself discusses a randomized **NC** algorithm for the same. Tzeng, in his another paper, considers the path equivalence probelm for non-deterministic automata which asks, given non-deterministic automata A and B, whether each word has the same number of accepting paths in A as in B. He gives a deterministic **NC** algorithm for deciding path equivalence which can be directly adapted to yield an **NC** algorithm for equivalence of probabilistic automata. However, this algorithm differs from Tzeng's polynomial time algorithm for the equivalence in the sense that it cannot output a word on which the two automata differ in case of inequivalence.

We now discuss some preliminaries which will help understand the algorithms for equivalence of probabilistic automata. The definitions and concepts in this section are taken from [KMO+12].

3 Preliminaries

3.1 Complexity Class NC

The complexity class $\mathbf{NC}^k$ for $k \geq 1$ is the subclass of P comprising those problems considered efficiently parallelisable. This class can be defined via parallel random access machines (PRAMs), which consist of a set of processors communicating through a shared memory.

Definition 3. *A problem is in $\boldsymbol{NC}^k$ for $k \geq 1$ if it can be solved in time $(\log n)^{\mathcal{O}(k)}$ (polylogarithmic time) on a PRAM with $n^{\mathcal{O}(1)}$ (polynomially many) processors.*

Problems in **NC** include directed reachability, computing the rank and determinant of an integer matrix, solving linear systems of equations and the tree isomorphism problem. It is not known whether $\mathbf{P} = \mathbf{NC}$.

Problems that are **P**-Hard under log-space reductions include circuit value and max-flow, and these are not in **NC** unless **P** = **NC**.

3.2 $\mathbb{Q}$-weighted automata

These automata are a generalization of Rabin's probabilistic automata (as we saw above).

Definition 4. *A $\mathbb{Q}$-weighted automaton $\mathcal{A} = (n, \Sigma, M, \alpha, \eta)$ consists of a positive integer $n \in \mathbb{N}$ representing the number of states, a finite alphabet Σ, a map $M : \Sigma \to \mathbb{Q}^{n \times n}$ assigning a transition matrix to each alphabet symbol, an initial (row) vector $\alpha \in \mathbb{Q}^n$, and a final (column) vector $\eta \in \mathbb{Q}^n$.*

Intuitively, this automaton associates with every transition upon the symbol $x \in \Sigma$ a rational number, which in some sense can be interpreted as the cost of that transition. Just like probabilistic automata, we can also associate a rational number with starting at a given state and one for ending at one. This way, probabilistic automata become a special case of $\mathbb{Q}$-weighted automata with the restriction that all weights must lie in the interval $[0, 1]$ and that $M(x)$ must be a stochastic matrix for all $x \in \Sigma$. Technically, since the probabilities need not be rational, this is not a strict special case. However, we can safely restrict our analysis to cases where the probabilities are only rational numbers. A more general case falls outside the scope of this paper.

We extend M to Σ^* as the matrix product $M(w_1 w_2 \dots w_n) := \prod_{i=1}^{n} M(w_i)$. The automaton $\mathcal{A}$ assigns to each word $\omega \in \Sigma^*$ a weight $\mathcal{A}(\omega) \in \mathbb{Q}$ where $\mathcal{A}(\omega) := \alpha M(\omega) \eta$. An automata $\mathcal{A}$ is said to be *zero* if $\mathcal{A}(\omega) = 0$ for all $\omega \in \Sigma^*$. With this view, we have the following definition.

Definition 5. *Two $\mathbb{Q}$-weighted automata $\mathcal{A}$ and $\mathcal{B}$ over the same alphabet Σ are said to be equivalent if $\mathcal{A}(\omega) = \mathcal{B}(\omega)$ for all $\omega \in \Sigma^*$.*

In the next section, we will see a randomized $\mathbf{NC}^2$ algorithm for deciding equivalence of $\mathbb{Q}$-weighted automata and in case of inequivalence, outputting a counterexample. This will form the basis for Tzeng's polynomial time algorithm [Tze92] for probabilistic automata in general.

4 Algorithms for language equivalence

We first describe the langauge equivalence problem for DFA using bisimulation and co-induction, as discussed in [HR15]. Recall that a deterministic finite automaton over the alphabet A is a triple (S, o, t) where S is a finite set of states, $o : S \to \{0, 1\}$ which determines if a state $x \in S$ is final ($o(x) = 1$) or not ($o(x) = 0$), and $t : S \to S^A$ is the transition function which returns, for each state $x \in S$ and for each letter $a \in A$, the next state $t_a(x)$. Thus, any DFA induces a function $[\![\cdot]\!]$ mapping states to formal languages $\mathcal{P}(A^*)$, defined by $[\![x]\!](\varepsilon) = o(x)$ for the empty word, and $[\![x]\!](a\omega) = [\![t_a(x)]\!](\omega)$ otherwise. For a state x, we call $[\![x]\!]$ the language accepted by x. Thus, the equivalence problem in this setting can be stated as: *given any two states $x_1, x_2 \in S$, is it the case that $[\![x_1]\!] = [\![x_2]\!]$?.*

4.1 Language equivalence via coinduction for deterministic finite automata

To understand Hopcraft's and Karp's algorithm for linear time equivalence of finite automata [HK71], we first need to understand the concept of bisimulation and progression.

Definition 6. *Given two relations $R, R' \subseteq S^2$ on states, we say that R progresses to R', denoted $R \rightarrowtail R'$, if whenever xRy then $o(x) = o(y)$ and for all $a \in A$, we have $t_a(x) R' t_a(y)$. A bisimulation is a relation R such that $R \rightarrowtail R$.*

Bisimulation, as expected, forms a sound and complete proof tecnique for checking language equivalence of DFA:

Proposition 1. ***(Coinduction)*** *Two states are language equivalent iff there exists a bisimulation that relates them.*

With these definitions in mind, a naive version of Hopcraft's and Karp's algorithm for checking language equivalence of the states x and y of a DFA (S, o, t) is given as follows. Basically, starting from x and y, this algorithm builds a relation R that, in case of success, is a bisimulation.

```
R ← empty;
todo ← empty;
Insert (x, y) in todo;
while todo is not empty do
    // Invariant is given as R ↣ R ∪ todo
    Extract (x', y') from todo;
    If (x', y') ∈ R then continue;
    If o(x') ≠ o(y') then return false;
    for a ∈ A do
        Insert (t_a(x'), t_a(y')) in todo;
    end
    Insert (x', y') in R;
end
Return true;
```

Algorithm 1: Naive algorithm for checking the equivalence of states x and y of a DFA (S, o, t).

This naive algorithm is quadratic: a new pair is added to R at each nontrivial iteration, and there are only n^2 such pairs, if there are n states in the DFA. To make this algorithm (almost) linear, Hopcraft and Karp actually record a set of equivalence classes rather than a set of visited pairs. As a consequence, their algorithm may stop earlier if it encounters a pair of states that is not already in R but belongs to its reflexive, transitive and symmetric closure. With this optimization, the produced relation R contains at most n pairs.

Having studied this basic algorithm for almost linear time checking of DFAs, we now discuss the algorithm in [KMO+12] which checks the equivalence of two $\mathbb{Q}$-weighted automata in randomized $\mathbf{NC}^2$ time.

4.2 Equivalence of $\mathbb{Q}$-weighted automata in randomized $\mathbf{NC}^2$ time

This approach is very closely related to Tzeng's algorithm for polynomial time comparison of probabilistic automata. Given two automata $\mathcal{B}$ and $\mathcal{C}$ that are to be checked for equivalence, one can compute an automaton $\mathcal{A}$ with $\mathcal{A}(w) = \mathcal{B}(w) - \mathcal{C}(w)$ for all $w \in \Sigma^*$. Then $\mathcal{A}$ is a zero automaton iff $\mathcal{B}$ and $\mathcal{C}$ are equivalent. Given $\mathcal{B} = (n^{(\mathcal{B})}, \Sigma, M^{(\mathcal{B})}, \alpha^{(\mathcal{B})}, \eta^{(\mathcal{B})})$ and $\mathcal{C} = (n^{(\mathcal{C})}, \Sigma, M^{(\mathcal{C})}, \alpha^{(\mathcal{C})}, \eta^{(\mathcal{C})})$, set $\mathcal{A} = (n, \Sigma, M, \alpha, \eta)$ with $n := n^{(\mathcal{B})} + n^{(\mathcal{C})}$ and

$$M(\sigma) := \begin{pmatrix} M^{(\mathcal{B})}(\sigma) & 0 \\ 0 & M^{(\mathcal{C})}(\sigma) \end{pmatrix}, \quad \alpha := \left(\alpha^{(\mathcal{B})}, -\alpha^{(\mathcal{C})}\right), \quad \eta := \begin{pmatrix} \eta^{(\mathcal{B})} \\ \eta^{(\mathcal{C})} \end{pmatrix}$$

Hence, we can now focus purely on the zeroness of $\mathcal{A}$, i.e. the problem of determining whether $\mathcal{A}$ is zero. This is not the same as saying that the language of $\mathcal{A}$ is empty, because in general, the transition weights can ne negative. However, in case of probabilistic automata, this analogy holds since probabilities are always non-negative. Thus, a witness word $w \in \Sigma^*$ against the zeroness of $\mathcal{A}$ is a witness against the equivalence of $\mathcal{B}$ and $\mathcal{C}$ as well. However, this does not mean that we have to check each of the infinitely many words in Σ^* to find such a witness. As it turns out, the following lemma, given by Tzeng in [Tze92] becomes crucial in this setting.

Lemma 1. *If $\mathcal{A}$ is not equal to the zero automaton, then there exists a word $u \in \Sigma^*$ of length atmost $n-1$ such that $\mathcal{A}(u) \neq 0$.*

The proof of this lemma becomes simple the moment we realize that given $\mathcal{A}$ has only n states, the largest word which traverses all these states exactly once will be of length $n-1$, and hence, if there is a word $v \in \Sigma^*$

such that $\mathcal{A}(v) \neq 0$ and $|v| > n-1$, then the Pigeonhole principle tells us that v must have traversed atleast one state, say q twice (or may be more than that). Hence, we can write $v = xyz$ where $x, y, z \in \Sigma^*$ and $|y| > 0$ such that x takes v from the start state to state q, then y loops around q and then z takes v from the state q to a final state where it gets accepted[2]. This means we can loop around q arbitrary number of times and still get accepted by the automaton, with possibly different output, and thus, xy^iz gets accepted by $\mathcal{A}$ for all $i \geq 0$. Then, setting $i = 0$ gives that xz is accepted by $\mathcal{A}$, the length of which is surely less than that of v. This inductive argument, hence, proves that if $\mathcal{A}$ has to accept a word outputting a rational number other than 0, then such a word can be found within the set of all words of length at most $n-1$. ■

Yet another lemma which is crucial to the develpoment of this randomized $\mathbf{NC}^2$ is called the Isolating lemma, as given by Mulmuley, Vazirani and Vazirani in [MVV87], primarily in the context of computing maximum matchings in graphs in $\mathbf{RNC}^3$. We state this lemma here without proof.

Lemma 2. *Let $\mathcal{F}$ be a family of subsets of a set $\{x_1, x_2, \ldots, x_N\}$. Suppose that each element x_i is assigned a weight w_i chosen independently and uniformly at random from $\{1, 2, \ldots, 2N\}$. Define the weight of $S \in \mathcal{F}$ to be $\sum_{x_i \in S} w_i$. Then the probability that there is a unique minimum weight set in $\mathcal{F}$ is at least $1/2$.*

Given these two lemmas, [KMO+12] formulates the **RNC** algorithm by introducting some more terminology. Given that the size of alphabet Σ is n, then for every $\sigma \in \Sigma$ and $1 \leq i \leq n$, choose a weight $w_{i,\sigma}$ independently and uniformly at random from the set $\{1, 2, \ldots, 2|\Sigma|n\}$. Then, we can define the weight of the word $u = \sigma_1\sigma_2 \ldots \sigma_n$ as

$$wt(u) = \sum_{i=1}^{n} w_{i,\sigma_i}$$

Note that this weight is different from $\mathcal{A}(u)$. After obtaining this weight, formulate a polynomial

$$P(x) = \sum_{k=0}^{n} \sum_{u \in \Sigma^k} \mathcal{A}(u) x^{wt(u)}$$

The intuition behind this approach is that the coefficient of x^m for any $m \geq 0$ is exactly the probability that $\mathcal{A}$ accepts a word whose weight is m. Thus, this polynomial is like a generating function for the sequence $p_1, p_2, \ldots$ where p_i denotes the probability that $\mathcal{A}$ accepts a string of weight i. Hence, for our problem of trying to see if $\mathcal{A}$ is the zero automaton, this polynomial must be identically zero, since the automaton in this case must accept with all words u with $\mathcal{A}(u) = 0$. However, if $\mathcal{A}$ is not the zero automaton, then Lemma 1 tells us that the set $\mathcal{F} = \{u \in \Sigma^{\leq n} \mid \mathcal{A}(u) \neq 0\}$ is not-empty. Thus there is a unique minimum weight word $u \in \mathcal{F}$, by Lemma 2 with probability atleast $1/2$. In this case, P contains the monomial $x^{wt(u)}$ with coefficient $\mathcal{A}(u)$ as it's smallest degree monomial. This implies $P \not\equiv 0$ with probability atleast $1/2$.

The last step to observe here is the formula

$$P(x) = \alpha \left(\sum_{i=0}^{n} \prod_{j=1}^{i} \sum_{\sigma \in \Sigma} M(\sigma) x^{w_{j,\sigma}} \right) \eta$$

and the fact that iterated products of matrices of univariate polynomials can be computed in $\mathbf{NC}^2$. This gives the required **RNC** algorithm for determining zeroness of weighted automata. Now, to extend this algorithm so that it also outputs a word u such that $\mathcal{A}(u) \neq 0$ in case $\mathcal{A}$ is non-zero, let the random choice of weights isolate a unique minimum-weight word $u = \sigma_1\sigma_2 \ldots \sigma_k$ such that $\mathcal{A}(u) \neq 0$. To detemrine whether $\sigma \in \Sigma$ is the i^{th} letter of u, we can increase the weight $w_{i,\sigma}$ by 1 while leaving all other weights unchanged and recompute the polynomial $P(x)$. Then σ is the i^{th} letter in u iff the minimum degree in P changes. All

[2]We say a word $u \in \Sigma^*$ is accepted by $\mathcal{A}$ if $\mathcal{A}(u) \neq 0$.
[3]Randomized **NC**

of these tests can be done independently, yielding an **RNC** operation.

The next subsection presents Tzeng's algorithm, which is very similar to the algorithm presented above. The only difference is in the running time. While Tzeng's algorithm takes biquadratic deterministic time, the above algorithm runs using randomization and parallelization in **RNC** time, which is a well-known subclass of $\mathcal{P}$, the class of all polynomial time solvable problems.

4.3 Language equivalence for probabilistic automata in polynomial time

This subsection discusses Tzeng's algorithm [Tze92], which compares the equivalence of two probabilistic automata with n_1 and n_2 states, respectively, in $\mathcal{O}((n_1 + n_2)^4)$ time and also returns the lexicographically minimum string on which they differ, in case of inequivalence. It can also be used to solve the path equivalence problem for NFA without ε-transitions and the equivalence problem for unambiguous finite automata in polynomial time. The paper also studies approximate equivalence (or δ-equivalence) problem for probabilistic automata by presenting an algorithm for the same. However, this algorithm terminates except for the case where the input δ is equal to the maximum difference of accepting probabilities of a string. For sake of brevity, the definitions and notations from the previous subsection are borrowed here.

Given two probabilistic automata $\mathcal{A}$ and $\mathcal{B}$, define for each string $x \in \Sigma^*$ the function $R(x) = \alpha M(x)$. Thus, for these two automata to be equivalent, we must have $R(x)\eta$ to be zero for all $x \in \Sigma^*$. To check this, let $H = \{R(x) \mid x \in \Sigma^*\}$. Then, if V represents the basis of $span(H)$ and $V \subseteq Null(\eta)$, we can say that the two automata are equivalent, since this would imply that for all $x \in \Sigma^*$, we have $\alpha M(x)\eta = 0$ which is exactly what we want. Now to convince ourselves that an algorithm as easy to understand as this one takes only polynomial time, note that Lemma 1 can be rephrased in this setting as follows:

Lemma 3. *Two automata with n_1 and n_2 states, respectively, are not equivalent iff there exists a string x of length atmost $n_1 + n_2 - 1$ such that $\alpha M(x)\eta \neq 0$.*

The proof of this lemma can be seen in [Paz14]. Using this, Tzeng formulates his algorithm by defining T to be a tree which has a node for every string in Σ^*. The root of this tree is $node(\varepsilon)$, the node for the empty string. Now, without loss of generality, assume that $\Sigma = \{0, 1\}$. A bigger alphabet will not make much difference in the nature of this algorithm. Then, every $node(x)$ in T has two children $node(x0)$ and $node(x1)$. Clearly, we can obtain $R(x0)$ as

$$R(x0) = R(x)M(0)$$

and similarly for $R(x1)$. The method then used to check equivalence is a breadth-first-traversal[4] over nodes in T, where at each node $node(x)$, we verify whether its associated vector $R(x)$ is linearly independent of the V (which is initially empty) that we have so far. If it is, we add it to V, otherwise we prune the subtree rooted at $node(x)$. We stop traversing T when every node in T is either visited or pruned. The vectors in the resulting set V will be linearly independent and form a basis for $span(H)$.

[4]The traversal order does not matter here. A different basis set may be generated, which will be equivalent to the one here. However, a breadth-first-traversal is necessary to output the lexicographically minimum string whose accepting probabilities are not the same by the two automata, in case of inequivalence.

Input : Two probabilistic automata $\mathcal{B}$ and $\mathcal{C}$
Set $V, N \leftarrow \emptyset$;
queue $\leftarrow node(\varepsilon)$;
while *queue is not empty* **do**
  Take an element $node(x)$ from the *queue*;
  if $R(x) \notin span(V)$ **then**
    Add $node(x0)$ and $node(x1)$ to the *queue*;
    Add $R(x)$ to V;
    Add $node(x)$ to N;
end
if $\forall v \in V.v\eta = 0$ **then**
  Return **yes**;
else
  Return $lex - \min\{x \mid node(x) \in N, R(x)\eta \neq 0\}$;

Algorithm 2: Tzeng's algorithm for checking equivalence of two probabilistic automata

The proof of correctness of this algorithm is given in [Tze92]. For the runtime complexity, assuming that arithmetic operations on rational numbers can be done in constant time, the total time taken by the algorithm is polynomial in the toal number of states.[5]

We now see how this algorithm can be extended to the case where the initial state probabilities α is not known for the probabilistic automata. Surprisingly, even for such a case, called the equivalence of uninitiated probabilistic automata, the algorithm runs in polynomial time.

4.4 Language equivalence for uninitiated probabilisitic automata

In this subsection, we show that the covering and equivalence problems for uninitiated probabilisitic automata are also polynomial-time solvable. A few definitions to get us going are as follows.

Definition 7. *1. An uninitiated probabilistic automaton U is a 4-tuple (S, Σ, M, F) where S is a finite set of n states, Σ is an input alphabet, M is the transition function as defined previously and $F \subseteq S$ is a set of final states. (Thus, given an initial state distribution ρ, we can form a probabilistic automaton $U(\rho)$ our of an uninitiated automaton U.)*

2. Let U_1 and U_2 be two uninitiated automata. Then U_1 is said to cover U_2 is for any initial-state distribution ρ_2 for U_1, there is an intitial-state distribution ρ_1 for U_1 such that $U_1(\rho_1)$ and $U_2(\rho_2)$ are equivalent.

3. Let U_1 and U_2 be two uninitiated probabilistic automata. Then U_1 and U_2 are said to be equivalent if U_1 covers U_2 and U_2 covers U_1.

Let $U_1 = (S_1, \Sigma, M_1, F_1)$ and $U_2 = (S_2, \Sigma, M_2, F_2)$ be two uninitiated probabilistic automata having n_1 and n_2 states, respectively. Let $Q_i(x) = M_i(x)\eta_F$. Then let J be an $(n_1 \times r)$-dimensional matrix whose column vectors $Q_1(x_1), Q_1(x_2), \ldots, Q_1(x_r)$ for some $r \leq n_1$ form a basis for the vector space $span(\{Q_1(x) \mid x \in \Sigma^*\})$. Let G be the similar matrix for Q_2. Then [Paz14] shows that if B is an $(n_2 \times n_1)$-dimensional stochastic matrix such that $BJ = G$, then U_1 covers U_2 iff for all $\sigma \in \Sigma$, we have $BM_1(\sigma) = M_2(\sigma)G$. The stochastic matrix B is a transformation from initial-state distributions for U_2 to those for U_1. This condition gaurantees that $U_2(\rho_2)$ and $U_1(\rho_2 B)$ are equivalent for any initial-state distribution ρ_2 for U_2. Since ρ_2 and B

[5]The set N contains $\mathcal{O}(n_1 + n_2)$ elements. The vector $R(x0)$, or $R(x1)$, can be calculated by multiplying $R(x)$ with $M(0)$, or $M(1)$ respectively, in $\mathcal{O}((n_1 + n_2)^2)$ time. To verify whether a set of $(n_1 + n_2)$-dimensional vectors is linearly independent needs $\mathcal{O}((n_1 + n_2)^3)$. Thus, the total time is $\mathcal{O}((n_1 + n_2)^4)$.

are stochastic, so is $\rho_2 B$. The problem of finding this matrix B can be reduced to the linear programming problem because of the restriction of stochasticity on B. If no such B exists then U_1 does not cover U_2. Once B has been found, it is easy to verify the condition. Thus, there is a polynomial time algorithm that takes as input two uninitiated probabilistic automata and determines whether one covers the other and also, if the two are equivalent.

Now, having studied the problem of exact equivalence and some great algorithms for this, let us now study the notion of approximate equivalence of probabilistic automata.

5 Approximate equivalence of probabilistic automata

In this section, we consider the problem of determining whether two probabilistic automata are δ-equivalent.

Definition 8. *For $\delta \geq 0$, two probabilistic automata are δ-equivalent if for every string $x \in \Sigma^*$ the probabilities that the two automata accept x differ by at most δ.*

Thus, two equivalent automata are zero-equivalent and any two probabilistic automata are 1-equivalent. However, the precise complexity class of such an equivalence is unknown, a restriction of this problem to positive definite stochastic transition matrices can be decided in polynomial time by a simple extension of Tzeng's algorithm. Without going into details, the main results of the paper [Tze92] in this regard are that there exists an algorithm that takes as inputs two probabilistic automata with this restriction along with a number $0 < \epsilon \leq 1$ and outputs a number δ such that δ is atmost the maximum difference of accepting probabilities for any string x. However, the algorithm takes exponential time. The same algorithm also checks for δ-equivalence in the same time.

However, if one wants to approximate this value of δ for probabilistic automata in general, the available options are based on similarity distances between the probability distributions simulated by the automata. Two such measures are the Kullback-Leibler divergence measure, as defined in [NS04], and the L_p distance, as defined in [CMR07].

Definition 9. *If $\mathcal{A}$ and $\mathcal{B}$ are two probabilistic sutomata, then the Kullback-Leibler divergence (or relative entropy) between $\mathcal{A}$ and $\mathcal{B}$ is denoted $D(\mathcal{A}\|\mathcal{B})$, and is defined as:*

$$D(\mathcal{A}\|\mathcal{B}) = \sum_{x \in \Sigma^*} \mathcal{A}(x) \log_2 \left(\frac{\mathcal{A}(x)}{\mathcal{B}(x)} \right)$$

The divergence can be interpreted as the additonal number of bits needed to encode data generated from $\mathcal{A}$ when the optimal code is chosen according to $\mathcal{B}$ instead. Although the sum is over all strings in Σ^*, this measure can be only computed on the lexicographically minimum string on which $\mathcal{A}$ and $\mathcal{B}$ differ, as reported by Tzeng's algorithm, and used to compute an appropriate δ. This value will only be an approximate value for δ, but it can be convertd into a better bound by taking the sum over the divergence on all strings that have length at most $n_1 + n_2 - 1$ and on which the two automata disagree. However, the problem in general is PSPACE-Complete.

Similarly, we can compute the L_p-distance between two probabilistic automata. The paper [CMR07] gives efficient exact and approximate algorithms for computing these distances for even p and proves that the problem is NP-Hard for all odd values of p using a reduction from Max-Clique. For unambiguous automata, this paper also shows that the L_{2p} distance between $\mathcal{A}$ and $\mathcal{B}$ can be computed in $\mathcal{O}(2p|\mathcal{A}|^3|\mathcal{B})|^3$ time exactly. If we remove this restriction of unambiguity, another theorem in the paper shows that this distance can be computed in $\mathcal{O}((|\mathcal{A}| + |\mathcal{B}|)^{6p})$ time.

6 Conclusion

This report presents a study on the notion of equivalence of probabilistic automata using a randomized $\mathbf{NC}^2$ algorithm and a polynomial time algorithm by Tzeng. Some side cases like unambiguous automata, approximate equivalence and the Isolating lemma are also presented. Although the problem of equivalence of non-deterministic finite state automata is PSPACE-Complete in general, it boils down to a polynomial time case for probabilistic automata using simple techniques from linear algebra. Finally, our limited knowledge of the complexity of approximate equivalence was shed some light upon by a polytime algorithm for computing L_{2p} distance between probability distributions, and hence, automata.

References

[AW92] Naoki Abe and Manfred K Warmuth. On the computational complexity of approximating distributions by probabilistic automata. *Machine Learning*, 9(2-3):205–260, 1992.

[CMR07] Corinna Cortes, Mehryar Mohri, and Ashish Rastogi. Lp distance and equivalence of probabilistic automata. *International Journal of Foundations of Computer Science*, 18(04):761–779, 2007.

[HK71] John E Hopcroft and Richard M Karp. A linear algorithm for testing equivalence of finite automata. Technical report, Cornell University, 1971.

[HR15] Thomas A. Henzinger and Jean-François Raskin. The equivalence problem for finite automata: Technical perspective. *Commun. ACM*, 58(2):86–86, January 2015.

[KMO+12] Stefan Kiefer, Andrzej S Murawski, Joël Ouaknine, Björn Wachter, and James Worrell. On the complexity of the equivalence problem for probabilistic automata. In *Foundations of Software Science and Computational Structures*, pages 467–481. Springer, 2012.

[MVV87] Ketan Mulmuley, Umesh V Vazirani, and Vijay V Vazirani. Matching is as easy as matrix inversion. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 345–354. ACM, 1987.

[NS04] Mark-Jan Nederhof and Giorgio Satta. Kullback-leibler distance between probabilistic context-free grammars and probabilistic finite automata. In *Proceedings of the 20th international conference on Computational Linguistics*, page 71. Association for Computational Linguistics, 2004.

[Paz14] Azaria Paz. *Introduction to probabilistic automata*. Academic Press, 2014.

[Tze92] Wen-Guey Tzeng. A polynomial-time algorithm for the equivalence of probabilistic automata. *SIAM Journal on Computing*, 21(2):216–227, 1992.